Comparative Analysis of Pre-trained Code Language Models for Automated Program Repair via Code Infill Generation

Iman Hemati Moghadam
i.hemati.moghadam@tue.nl
Eindhoven University of Technology
Eindhoven, The Netherlands

Oebele Lijzenga
oebelelijzenga@gmail.com
University of Twente
Enschede, The Netherlands

Vadim Zaytsev
vadim@grammarware.net
University of Twente
Enschede, The Netherlands

Abstract

Automated Program Repair (APR) has advanced significantly with the emergence of pre-trained Code Language Models (CLMs), enabling the generation of high-quality patches. However, selecting the most suitable CLM for APR remains challenging due to a range of factors, including *accuracy*, *efficiency*, and *scalability*, among others. These factors are interdependent and interact in complex ways, making the selection of a CLM for APR a multifaceted problem.

This study systematically evaluates 20 pre-trained CLMs, ranging from 60M to 16B parameters, on the HumanEval-Java benchmark (163 buggy Java methods). The evaluation examines bug-fixing accuracy, resource consumption, compilability, patch diversity, and sampling strategies (beam search vs. nucleus sampling).

Results indicate that larger models such as CodeLLaMA-13B and StarCoder generally perform better in bug fixing and compiler error handling, but scale alone does not guarantee effectiveness, as some (e.g., CodeGen2) perform poorly despite their size. Notably, memory usage increases with model size, but time consumption does not exhibit a clear correlation, suggesting that efficiency is influenced by architecture rather than scale alone. Additionally, nucleus sampling slightly outperforms beam search, though the difference is not statistically significant. Since no single CLM fixes all bugs, these findings highlight the potential of hybrid or ensemble-based CLM-driven APR approaches for more robust bug-fixing.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Keywords: Automated program repair, pre-trained code language model, empirical study, zero-shot learning, Java

GPCE '25, Bergen, Norway

ACM Reference Format:
Iman Hemati Moghadam, Oebele Lijzenga, and Vadim Zaytsev. 2025. Comparative Analysis of Pre-trained Code Language Models for Automated Program Repair via Code Infill Generation. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '25), July 3–4, 2025, Bergen, Norway*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3742876.3742881

1 Introduction

Precise and efficient bug fixing is essential for ensuring the quality and reliability of software. Improving bug-fixing methods not only enhances overall software quality but also drives advancements in related fields, including intelligent code recommendations, accurate program synthesis, and efficient product-line development.

Automated Program Repair (APR) has been an active research area, with various techniques developed to address software bugs efficiently [25, 31, 36, 37, 52, 53, 60, 66, 73]. These techniques can be categorized into constraint-based, search-based, template-based, and learning-based approaches. Among them, learning-based approaches, particularly pre-trained Code Language Models (CLMs), have emerged as a promising solution for generating high-quality patches, showcasing the potential of generative AI in APR.

Supervised learning-based APR techniques leverage machine learning to detect and generalize bug-fixing patterns from labeled code. These techniques often outperform traditional rule-based methods in accuracy and the diversity of generated patches [64, 73]. However, their effectiveness depends heavily on the quality and diversity of the training data and can be hindered by issues such as overfitting [27] and limited generalization. In contrast, CLMs address these limitations by learning from large-scale, unlabeled code corpora across multiple programming languages. Unlike conventional learned-based APR models, CLMs capture broader coding patterns and semantics, often outperforming earlier APR techniques even without fine-tuning for the task of program repair [70, 76].

Despite their potential, effectively applying CLMs to APR comes with significant challenges. Factors such as model

size, training data, and learning strategies (zero-shot vs. fine-tuning) substantially affect repair performance and computational costs [35, 43, 69, 75]. Moreover, additional complexities arise in ensuring code compilability, achieving diverse fixes, and managing different sampling strategies. While prior studies [35, 43, 68, 69, 75] have explored CLM-based APR, they have generally evaluated a limited number of models—typically between five and ten—leaving open questions about how model size, architecture, and sampling techniques impact bug-fixing performance, as well as the trade-offs between efficiency, cost, and effectiveness. Consequently, researchers and practitioners face challenges in selecting the most suitable CLM for APR. Therefore, each new APR technique should be evaluated across multiple CLMs to assess its performance under different conditions.

This study aims to address these gaps by conducting a comprehensive evaluation of 20 pre-trained CLMs, ranging from 60 million to 16 billion parameters, for automated program repair. Leveraging the HumanEval-Java benchmark [43], which consists of 163 buggy Java methods, we provide an in-depth analysis of CLM performance across several key dimensions. Following a widely adopted evaluation protocol [43, 69, 70, 76], we *mask* buggy lines of code and task the models with generating *accurate* code infills. Through this large-scale analysis, we explore the relationship between model size, repair quality, and computational efficiency, as quantified by bug-fixing accuracy, compilation success rates, and resource consumption, while also assessing how different sampling strategies (beam search vs. nucleus sampling) affect patch quality and diversity.

Our study extends prior research on CLM-based APR [35, 43, 68, 69, 75] by incorporating newer state-of-the-art models, such as StarCoder [49], SantaCoder [4], CodeLLaMA-Instruct [61], and Refact [5], which have not been previously assessed. Another key novelty of this work lies in its dual focus on patch diversity and sampling strategies. Beyond assessing fix accuracy, we examine how different sampling methods influence repair outcomes and analyze the complementarity and overlap of patches generated by various CLMs. This provides deeper insights into the diversity of solutions produced, shedding light on the strengths and limitations of different models in generating robust and effective bug fixes.

In summary, the contributions of our study are as follows:

1. Comprehensive CLM Evaluation – We systematically evaluate 20 pre-trained CLMs on the HumanEval-Java benchmark, assessing their ability to generate accurate code infills for APR.
2. In-Depth Performance Analysis – We compare CLMs across various metrics, including bug-fixing effectiveness, compilation rates, resource usage, sampling strategies, and fix diversity, providing insights into model strengths and weaknesses.
3. Public Dataset & Reproducibility – We release all experimental data and code [1], fostering reproducibility and future research on CLM-based APR.

The remainder of this paper is structured as follows: Section 2 reviews related work on CLMs and APR. Section 3 describes the methodology, including CLM selection and experimental setup. Section 4 presents the results, followed by threats to validity in Section 5. Finally, Section 6 provides conclusions and discusses future research directions.

2 Related Work

Automated Program Repair (APR) generally employs a generate-and-validate paradigm—where candidate patches are automatically produced and then verified against test suites—to ensure correctness. This approach can be classified into search-based [48, 72], template-based [46, 51], and learning-based methods [44, 70]. Search-based methods explore the space of possible code modifications to automatically generate candidate patches, yet they are often susceptible to overfitting [36]. Template-based techniques improve efficiency by leveraging predefined fix patterns; however, their applicability is limited by the diversity of available templates [69, 76]. In contrast, learning-based strategies utilize deep learning models to offer a wider range of patch candidates, though they are challenged by dataset dependencies and significant training costs [43, 64, 80].

In recent years, pre-trained CLMs have emerged as a transformative advancement in APR, utilizing vast, unlabeled code datasets to enable more comprehensive bug-fixing capabilities [43, 69]. While CLMs generally outperform traditional methods, their effectiveness is influenced by several interconnected factors, making their optimization for APR a complex challenge. Further investigation is needed into intrinsic model factors—such as model-specific hyperparameters, size, and architecture—and extrinsic optimization techniques, including fine-tuning, sampling strategies, and prompt engineering. Equally important is a comprehensive evaluation of performance metrics like bug-fixing accuracy, patch diversity, and resource consumption to better understand the trade-offs and interactions among these factors.

To better understand these complexities, recent studies have compared the performance of various CLMs for automated program repair. For instance, Jiang et al.[43] evaluated the bug-fixing capabilities of 10 CLMs, Xia et al.[69] compared 9 CLMs, and Huang et al.[35] analyzed 5 CLMs. Similarly, Zhang et al.[75] and Wu et al. [68] examined the effectiveness of 5 CLMs in addressing security vulnerabilities. In the following, we summarize the key findings from these studies, highlighting their contributions and limitations.

Fine-Tuning Effects: In program repair, fine-tuning involves training a model on pairs of buggy code and their fixes [35]. While some studies [35, 43, 56, 69, 70, 75] show that existing CLMs can perform well without fine-tuning—known

as zero-shot learning, where models are applied directly without task-specific training—others highlight significant benefits from it [35, 43, 68, 75]. For instance, Jiang et al. [43] found that fine-tuning can substantially improve the accuracy of the used CLMs, with improvements of at least 31%. However, there are still numerous aspects yet to be explored. For example, while Zhang et al. [75] found that increasing the training dataset enhances software vulnerability detection, Jiang et al. [43] observed that excessive fine-tuning data can reduce bug-fixing efficacy. Furthermore, Huang et al. [35] demonstrated how different ways to represent training data may affect the repair capability of CLMs. Jiang et al. [43] also highlighted the risk of introducing bias when the fine-tuning dataset is not carefully designed, potentially causing the model to favor certain bug types over others.

Prompt Selection's Impact: In program repair, a prompt can be at least a buggy program with extra information like bug location, context surrounding the buggy code, relevant error messages, and expected behaviour. These details provide more guidance to the model in its repair process. However, varying characteristics of CLMs prevent the creation of a universally effective prompt. For example, Jiang et al. [43] found that including buggy lines in the prompt resulted in 6% to 78% fewer fixed bugs, although some unique bugs were resolved. Conversely, Wu et al. [68] observed that including buggy lines in the prompt helped InCoder [29] fix more bugs. Zhang et al. [75] observed providing the context surrounding the buggy code while improving the accuracy of some CLMs, yielding negative effects on other models (e.g., CodeBERT [28] and UniXcoder [32]). In another study, Xia et al. [69] observed that providing context after the buggy line can significantly improve the accuracy, particularly for Codex [7], and led to more successful correcting as well as compiling patches. In a related study [50], we found that increasing context size in CodeLLaMA-13B [61] only improves performance up to a point. While adding more context could theoretically enhance CLM performance by exposing more code patterns and symbols, we observed that the optimal performance was achieved with approximately half of the available context size (i.e., 400 code lines). Additionally, larger context sizes lead to longer execution times.

Model Size vs. Bug-Fixing Performance: Previous studies, such as Jiang et al. [43], and Xia et al. [69], along with findings of this study, highlight that larger models tend to exhibit better bug-fixing performance compared to smaller models (e.g., CodeGen-2B [55] vs. InCode-6B [29]). However, this pattern is not observed across all models and studies. For example, Xia et al. [69] observed that the Codex-12B [7] outperforms the largest employed model, GPT-NeoX-20B [6]. In another study, Huang et al. [35] found that the smallest employed model, UniXcoder-125M [32], matched or outperformed the largest employed model, CodeT5-base-220M [67], contrasting with the results of Zhang et al. [75]. While these results confirm that not necessarily large CLMs have better bug-fixing capability, it is important to recognise that various other factors may yield contrasting results in different studies. For example, variations in beam search and nucleus sampling, or the values assigned to the internal parameters of models, such as temperature, can impact bug-fixing outcomes, as observed in our study and others [30, 75].

Patch Compilability: Research shows that a CLM's ability to generate compilable patches depends on several factors, including model size, fine-tuning, and code context. Larger models typically produce patches with fewer syntactic and semantic errors [43, 69]. However, Jiang et al.[43] observed that before fine-tuning, CodeT5 [67] often generated syntax errors, even more frequently than traditional machine learning approaches. Additionally, Xia et al. [69] found that providing both prefix and suffix code around the buggy line significantly improves a model's ability to generate syntactically valid patches by improving contextual understanding. In contrast, omitting the suffix code increases the likelihood of syntax errors.

3 Methodology

This section describes the methodology employed in this work to evaluate the performance of several pre-trained CLMs. First, an overview of the proposed approach is presented, and then the criteria for the models' inclusion and exclusion are considered. Next, we present the dataset used to evaluate model performance and discuss implementation aspects, including experimental design.

3.1 Ovierview of the Proposed Approach

Fig. 1 illustrates the automated program repair process employed in this study. The process consists of several phases, where the process starts with feeding a buggy program and a set of test cases for which at least one test fails, indicating the presence of a bug. In the first step, *Bug Localization*, the algorithm uses fault localization techniques to identify the suspicious code location, pinpointing the lines likely to contain the bug. Next, in *Line Masking*, the identified buggy line is masked by replacing the suspicious segment with a placeholder, preparing it for potential fixes. In the following phase, *CLM Patch Generation*, the CLM is configured based on developer settings to suggest multiple possible fixes for the masked code. After generating patches, the next phase, *Patch Validation*, involves running all generated patches through the set of test cases. Those patches that pass all test cases are considered plausible. Finally, in the *Patch Selection* phase, the plausible patches are manually evaluated to identify those that effectively rectify the bug. The selected patch is then applied to the program as a corrective solution.

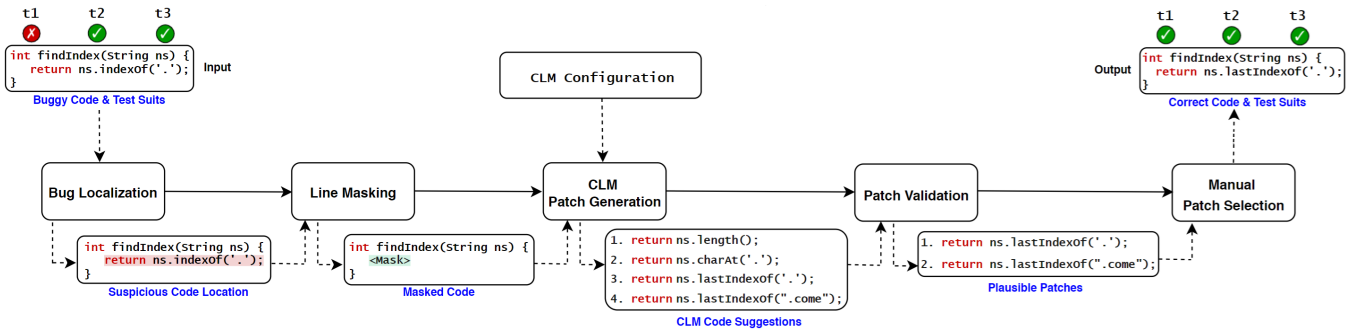


Figure 1. Automated Program Repair via Code Infill Generation.

3.2 Selected Pre-trained CLMs

CLMs are identified through an informal snowballed exploration of academic papers (e.g., [35, 43, 68, 69, 75]), surveys (e.g., [26, 33, 34, 66, 74, 77, 79, 81]) and online resources (e.g., [38]), guided by the following **four** inclusion and exclusion criteria.

First, CLMs must support *mask prediction*, meaning they can generate code completions or infills within a predefined prompt template that contains masked tokens. This process should occur *without prompt engineering*—i.e., without tailoring the prompt structure to a specific task—and in a *zero-shot setting*, where the model performs the task without any task-specific fine-tuning or in-context learning from example demonstrations. Thus, CLMs like GPT-NeoX [6], Mixtral [42] and CodeGeeX2 [78] are excluded.

Second, CLMs must be publicly available and capable of being *executed locally* to facilitate the acquisition of cost metrics and the configuration of sampling methods. Thus, CLMs like GPT-4 [2] and Codex [7] are excluded. We also exclude models that showed weak performance in previous studies (e.g., [35, 43, 68, 69, 75]).

A total of 25 CLMs were initially selected based on the three key inclusion criteria outlined above. These models, which vary in size and configuration, are presented in Table 1, with parameter sizes expressed in millions (M) or billions (B). Models highlighted in grey were subsequently excluded due to performance shortcomings or other limitations identified during experimentation, corresponding to the fourth criterion discussed in the following paragraph.

During our experiments, we found that CodeGen-2 [54] consistently outperformed earlier versions of CodeGen [55], leading us to exclude the earlier versions from our study. CodeBERT [28] was also excluded, although it shows promising results in AlphaRepair [70]. CodeBERT predicts one token per masked line, so the number of tokens to generate a masked line must be guessed beforehand. AlphaRepair addresses this by generating lines of varying lengths, up to $L + 10$ tokens, where L is the number of tokens in the original line. However, this approach is significantly slower than UniXCoder [32], which is capable of generating an arbitrary number of tokens for a single mask token. Since CodeBERT is outperformed by UniXCoder [32], which has similar size and performance characteristics, it was reasonable to exclude CodeBERT from our evaluation.

Table 1. Overview of the CLMs considered for evaluation. Model variants are distinguished by their parameter sizes, expressed in millions (M) or billions (B). Grey-shaded entries indicate models excluded due to performance issues.

Name	Variants
CodeBERT [28]	125M [8]
CodeGen [55]	350M [11], 2B [10], 6B [12], 16B [9]
CodeGen-2 [54]	1B [14], 3.7B [15], 7B [16], 16B [13]
CodeLLaMA [61]	7B [19], 13B [17], 7B-instruct [20], 13B-instruct [18]
CodeShell [71]	7B [21]
CodeT5 [67]	small (60M) [24], base (220M) [22], large (737M) [23]
InCoder [29]	1B [40], 6B [41]
PLBART [3]	base (140M) [57], large (400M) [58]
Refact [47]	1.6B [59]
SantaCoder [4]	1.1B [62]
StarCoder [49]	15.5B [63]
UniXCoder [32]	125M [65]

3.3 Employed Dataset

We employ the HumanEval-Java dataset [43], an APR benchmark adapted from the original HumanEval dataset [7]. It comprises 163 single-hunk Java bugs, manually injected to span a range of complexity, from simple issues such as incorrect operator usage to more intricate logical errors requiring multi-line modifications. As the dataset is systematically derived from HumanEval [7] and constructed to prevent overlap with training data, it mitigates data leakage and ensures that pre-trained models have not been exposed to these samples. Each bug is accompanied by its corrected version and a test suite, containing an average of 6.3 test cases per infill task to evaluate solution correctness [43].

3.4 Experiment Design

The evaluation of CLMs' code infilling capabilities is conducted using a **single-line code infilling** approach, where the buggy line is masked, and a CLM is used to generate code to fill in the masked line. Note that we omit the *Fault Localization* phase, shown in Fig. 1, and directly mask the buggy line to avoid the inefficiencies of fault localisation on CLM accuracy.

We evaluate CLMs using two sampling techniques: *beam search* and *nucleus sampling*. Beam search generates token sequences by exploring several candidates at each step and keeping only the best ones. The *beam size* specifies how many sequences are considered at once. A larger beam size improves result quality but increases memory usage and computation time. In contrast, nucleus sampling is a probabilistic method that generates token sequences by selecting tokens from a subset of the most likely options. At each step, it picks tokens from the top_ p percentage of the probability distribution. Tokens are chosen randomly, with more likely tokens being selected more often. The *temperature* parameter adjusts randomness: temperatures between 0 and 1 make the output more predictable, temperatures above 1 increase randomness, and a value of 1 preserves the original probability distribution.

Sampling techniques are crucial in APR because they directly influence the diversity and quality of generated patches. Beam search explores multiple high-probability solutions, making it effective for identifying promising fixes. In contrast, nucleus sampling introduces stochasticity, which enhances diversity by generating a broader range of candidate patches. This diversity is particularly beneficial in scenarios where multiple plausible fixes exist or where deterministic methods may overlook viable alternatives.

In this study, beam search is evaluated with beam sizes 5 and 10, and nucleus sampling is evaluated with all combinations of top_p $\in \{0.2, 0.4, 0.6, 0.8\}$ and temperature $\in \{0.1, 0.4, 0.7, 1.0, 1.3, 1.6, 1.9\}$. These parameter ranges are selected based on the results of informal preliminary testing, which showed no benefit in evaluating top_p = 1.0, or temperature ≥ 2.0. Moreover, beam size ≥ 10 led to excessive VRAM usage without demonstrating noticeably improved results to justify the cost. Thus, a total of 2 and 28 sampling configurations are evaluated for beam search and nucleus sampling, respectively. It should be noted that *UniXCoder* and *Refact* were only evaluated using beam search and nucleus sampling, respectively, as only these sampling methods are provided out-of-the-box for these models.

Each CLM is applied to the single-line infilling benchmark under each sampling configuration in a zero-shot setting, with one run per configuration. For each infill task, *five* completions are generated, and each is evaluated by compiling and testing the resulting code.

3.5 Experimental Hardware Setup

All experiments were performed on machines equipped with a 2.1 GHz Intel Xeon Silver 4216 processor, 32 GB of memory, and a single NVidia A40 GPU with 48 GB of VRAM. Although the CPU was shared with other users, 16 physical cores are allocated for up to 15 parallel patch evaluations to prevent starvation of CPU capacity. The GPU, however, was exclusively dedicated to the APR process and was not shared with other users. For experiments, a 16-bit quantization was used to manage VRAM usage effectively.

3.6 Research Questions

To guide our evaluation and analysis, we aim to address the following research questions:

- **RQ1:** How do different CLMs perform in bug fixing using a single-line infilling approach? This question is divided into four sub-questions:
  - **RQ1.1:** How do beam search and nucleus sampling compare in CLM bug fixing?
  - **RQ1.2:** What is the relationship between CLM size and bug-fixing capabilities?
  - **RQ1.3:** How does the success rate of the infilling task differ between a single attempt ($N = 1$) and when at least one of five attempts ($N = 5$) is correct?
  - **RQ1.4:** How does CLM size affect the compilation rate?
- **RQ2:** How do the memory and time usage of different CLMs correlate with their bug-fixing performance?
- **RQ3:** How does code infill diversity vary across CLMs, and what are the trade-offs with correctness and performance?

4 Experiment Results

The performance of CLMs is evaluated using two key metrics. The $N = 1$ metric measures the number of infill tasks where the first generated infill is test-adequate, meaning the code compiles and passes all test cases. The $N = 5$ metric counts the number of infill tasks where any of the first five generated infills are test-adequate. These metrics assess both the model's ability to generate correct fixes on the first attempt and its potential to produce a valid fix within multiple trials.

Table 2 compares these metrics, highlighting variations in bug-fixing performance across models. A small *difference* between $N = 1$, and $N = 5$ suggests a model generates high-quality fixes early, while a larger gap indicates reliance on multiple attempts to find a correct solution. These differences, observed across models, reflect the impact of sampling strategies and repair tendencies.

The results presented in Table 2 are based on the best configuration for each model, whether it is beam search or nucleus sampling. A comprehensive set of results for all models and configurations can be found in [1].

Table 2. Success rates for infill tasks for the first ($N = 1$) and first five results ($N = 5$), along with resource usage, sampling settings, and compilation rate for each model at optimal configuration. Masked time and VRAM are reported for $N = 1$.

CLM	Size	Fixes N=1	Fixes N=5	Difference	Mask Time	VRAM (GB)	Beam Size	top_p	Temperature	Compilation Rate
UniXCoder	125M	26	34	+8 (30.8%)	107.3	4.6	5			46.3%
Refact	1.6B	112	121	+9 (8.0%)	107.6	5.6		0.6	0.1	95.3%
SantaCoder	1.1B	101	107	+6 (5.9%)	153.1	3.9		0.6	0.7	97.1%
CodeShell	7B	125	132	+7 (5.6%)	279.3	18.9		0.6	0.7	96.1%
StarCoder	15.5B	136	141	+5 (3.7%)	307.1	33		0.8	0.7	97.8%
PLBART Base	140M	41	67	+26 (63.4%)	417.3	1.4	10			70.1%
PLBART Large	400M	42	76	+34 **(81.0%)**	746.4	2.8	10			80.3%
CodeT5 Small	60M	22	38	+16 (72.7%)	**34.7**	**1**	10			31.3%
CodeT5 Base	222M	50	61	+11 (22.0%)	57.8	1.5	5			46.9%
CodeT5 Large	737M	69	69	—	242.5	2.8		0.4	0.1	94.4%
CodeLLaMA 7B	7B	126	135	+9 (7.1%)	303.1	23.4		0.6	1	98.0%
CodeLLaMA 13B	13B	**137**	**143**	+6 (4.4%)	255.5	34.8		0.6	0.7	**98.6%**
CodeLLaMA 7B Instruct	7B	74	116	+42 (56.8%)	1103.7	26.5		0.8	0.7	62.1%
CodeLLaMA 13B Instruct	13B	76	132	**+56** (73.7%)	1312.6	43.3		0.8	1.3	62.0%
CodeGen2 1B	1B	2	3	+1 (50.0%)	618.7	44.3		0.4	1.3	1.7%
CodeGen2 3.7B	3.7B	8	9	+1 (12.5%)	490.5	25.4		0.4	1.3	36.8%
CodeGen2 7B	7B	44	46	+2 (4.5%)	1510.7	41.2		0.6	0.4	48.0%
CodeGen2 16B	16B	124	126	+2 (1.6%)	1429.5	44.3		0.8	0.7	82.0%
InCoder 1B	1B	63	74	+11 (17.5%)	187.5	6.2	5			54.7%
InCoder 6B	6B	70	82	+12 (17.1%)	170.4	21.5	5			63.1%

For the remainder of this study, we focus on $N = 1$ performance for evaluating metrics (e.g., memory usage, time consumption, and compilation rate), based on the assumption that it is most relevant to the proposed APR technique, especially given the resource and time constraints of CLMs. Instances where $N = 5$ performance is used are explicitly noted.

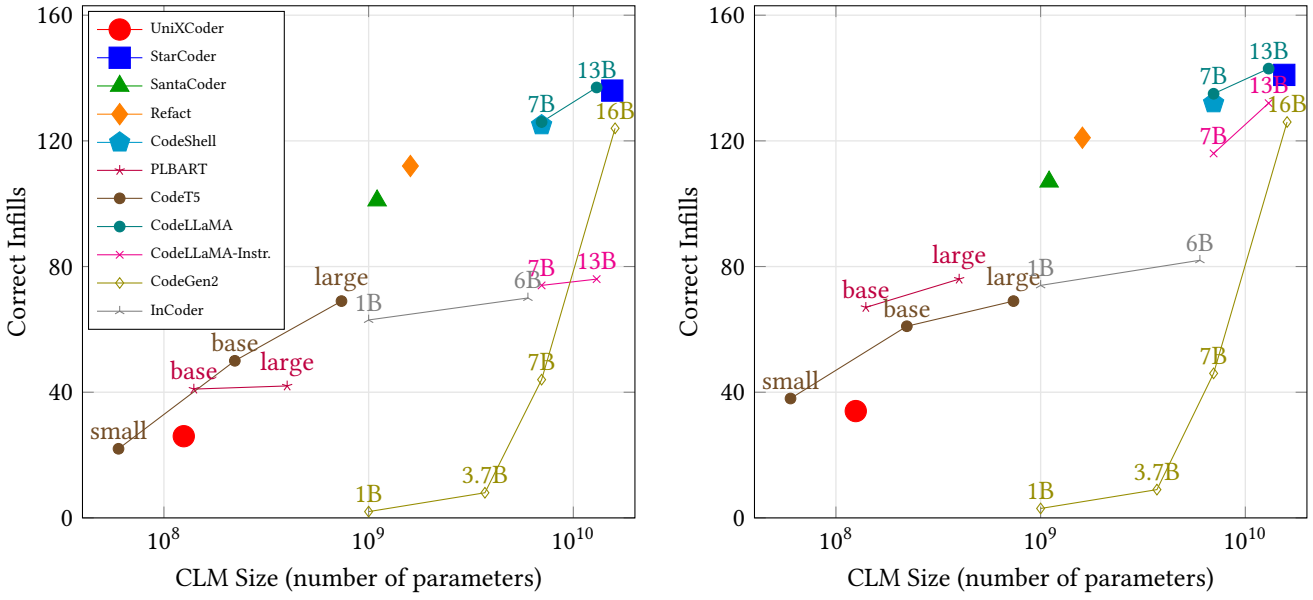
4.1 RQ1: Evaluating CLM Bug Fixing

4.1.1 Results for RQ1.1. This research question explores how beam search and nucleus sampling compare in their effectiveness for CLM bug fixing. Table 2 presents the best performance achieved by each model across different sampling settings, including beam search and nucleus sampling. As shown, for similar models of varying sizes, such as CodeLLaMA, the same sampling technique yields the best results. The exception is CodeT5, where CodeT5-Large performs better with nucleus sampling compared to beam search, which was more effective for the smaller CodeT5 models.

Overall, our experiments reveal that nucleus sampling generally outperforms beam search showing an average improvement of 13.6% for $N = 1$ and 3.7% for $N = 5$, *across all tested configurations*, rather than the best-performing configurations shown in Table 2. However, the Wilcoxon signed-rank test indicates that these differences are not statistically significant at $p < 0.05$, suggesting that neither sampling method (beam search or nucleus sampling) has a clear advantage.

We also compared the results for beam sizes of 5 and 10, and we found no significant difference in bug-fixing performance according to the Wilcoxon signed-rank test at $p < 0.05$. Therefore, the results do not exhibit a discernible advantage for either employed sampling size in beam search. However, when evaluating nucleus sampling, the performance gap between the *best* and *worst* configurations was notable. Specifically, we observed improvements of 29.3% for $N = 1$ and 23.7% for $N = 5$ in the best configurations compared to the worst ones, with Wilcoxon signed-rank test p-values of 0.0001816 and 0.0001426, respectively. This indicates that configuration in nucleus sampling can have a significant impact on results.

4.1.2 Results for RQ1.2. This research question explores the relationship among CLM size and their bug-fixing capabilities.



(a) Performance of various CLMs versus their size for $N = 1$.

(b) Performance of various CLMs versus their size for $N = 5$.

Figure 2. Comparison of CLM performance versus model size.

Fig. 2 shows a different view of Table 2, displaying the relationship between model size and the performance of the best results for each model. This figure reveals a wide range of size and performance points for both N = 1 and N = 5.

Focusing on Fig. 2a, CodeT5 the smallest model with 60 million parameters, generates 22 correct infills, outperforming CodeGen2 1B, and 3.7B, which produce 2 and 8 correct infills, respectively. Indeed, in terms of size versus performance, GodeGen2 performs the weakest, and even its best model (with 16B parameters), which is the largest model in our study, performs poorly compared to smaller models like CodeSell, StarCoder, and CodeLLaMA 7B and 13B. Overall, both preliminary evaluation and empirical evaluation demonstrate that the smaller CodeGen2 models (1B, 3.7B, and 7B) are ineffective for code-infilling tasks, and its largest model does not perform as good as other smaller models.

Focusing on the top-performing models, CodeLLaMA-13B and StarCoder demonstrate better results. For $N = 1$, they achieve notable results by fixing 137 and 136 bugs, respectively, and for $N = 5$, they further demonstrate their effectiveness by addressing 143 and 141 bugs out of 163, respectively.

As observed from Fig. 2, a strong positive correlation between size and performance, except for PLBART in $N = 1$, can be observed for all investigated CLMs, demonstrating that a higher cost can yield better performance. This strong positive correlation was confirmed by calculating the Spearman correlation coefficient for the *average performance across all configurations*, and not the best-performing configurations shown in Table 2, considering both beam search and nucleus sampling methods *separately* for $N = 1$ and $N = 5$.

We also observed that as model size increases, performance gains tend to diminish. For instance, SantaCoder, 1.8x larger than CodeT5 Small, achieves a 359% improvement, while StarCoder, 14x larger than SantaCoder, only shows a 35% gain for $N = 1$.

Another interesting observation is that CodeLLaMA-instruct 7B and 13B produce 74 and 76 correct infills, respectively, while the base CodeLLaMA model generates 126 and 137 correct infills. Despite both models sharing similar architecture, the CodeLLaMA-instruct models, which are fine-tuned for instruction-based tasks, show reduced mask prediction performance. This suggests that the instruction fine-tuning may have compromised their infilling capabilities compared to the base CodeLLaMA model. More details about this underlying issue will be provided when discussing RQ1.3.

4.1.3 Results for RQ1.3. This research question investigates how the success rate of the infilling task changes when considering the first infill ($N = 1$) versus at least one of five infills ($N = 5$) being correct. As indicated by the *Difference* field in Table 2 and demonstrated by comparing Fig. 2a with Fig. 2b, the performance of all CLMs—except for CodeT5-Large—improves with $N = 5$ compared to $N = 1$ for the best results. The Wilcoxon Signed-Rank Test confirms this improvement is statistically significant across all configurations and not just for the best results. On average, $N = 5$ achieves a 24.3% improvement over $N = 1$ *across all configurations*, with CodeGen2 16B showing the smallest gain at 1.9% and PLBART-Large the most at 60.4%. These findings

underscore the importance of generating and evaluating multiple candidate infills, which leads to statistically significant performance improvements across all configurations.

We further analyze *the best-performing configurations* and observe these insights. CodeLLaMA-Instruct 7B and 13B perform 56.8% and 73.7% better, respectively, for $N = 5$. An investigation of the generated infills shows that CodeLLaMA-Instruct generates high-quality infills similar to CodeLLaMA, but frequently fails to halt token generation at the appropriate moment, which yields infills containing superfluous code that disturbs the behaviour of the program. CodeLLaMA-Instruct performs better at $N = 5$ under high-diversity sampling configurations as it occasionally stops token generation at the appropriate moment, or generates superfluous code which does not disturb the behaviour of the program. In contrast, PLBART-Base and Large and CodeT5-Small perform 63.4%, 81%, and 72.7% better, respectively, for $N = 5$. In contrast to CodeLLaMA-Instruct, these CLMs do not fail to appropriately halt token generation. Instead, correct subsequent infills are frequently a slight variation of the first, incorrect infill. Note that PLBART-Base and Large and CodeT5-Small demonstrate the best performance for both $N = 1$ and $N = 5$ under beam search with a beam size of 10, which is likely to yield predictable but diverse infills.

4.1.4 Results for RQ1.4. Negative evaluations of code infills are mostly comprised of compilation errors and test failures. This research question investigates how the size of CLMs affects the *compilation rate*. We report the compilation rate for *each model* based on *all five* infills for the *163 infill tasks* using the below formula:

COMPILATION RATE = #COMPILABLE INFILLS / (163 * 5)

Fig. 3 shows the compilation rate of infills for the best result for each model plotted against the size of CLMs. Only seven of the 20 analysed pre-trained models produce upwards of 90% of compilable output. Indeed, infills generated by CodeT5-Large (737M), SantaCoder (1.1B) and Refact (1.6B) compile successfully 94.4% to 97.1% of the time. In contrast, compilation rates for larger models like CodeShell (7B), CodeLLaMA (7B and 13B) and StarCoder (15.5B) range from 96.1% to 98.6%. CodeLLaMa-13 B demonstrates the best performance among these models. Conversely, the weakest result was achieved by CodeGen2 (1B), where only 1.7% of generated patches were compilable, highlighting the main reason for this model's low accuracy.

Worth mentioning that while the compilation rate for some models, such as PLBART-Large with an 80.3% success rate, is not necessarily a good indicator of overall model performance, it is only 18.3% lower than the best model despite being 32.5 times smaller. This suggests that we can generate infills, for example, with a larger beam size using minimal resources and achieve similar results to those of larger models.

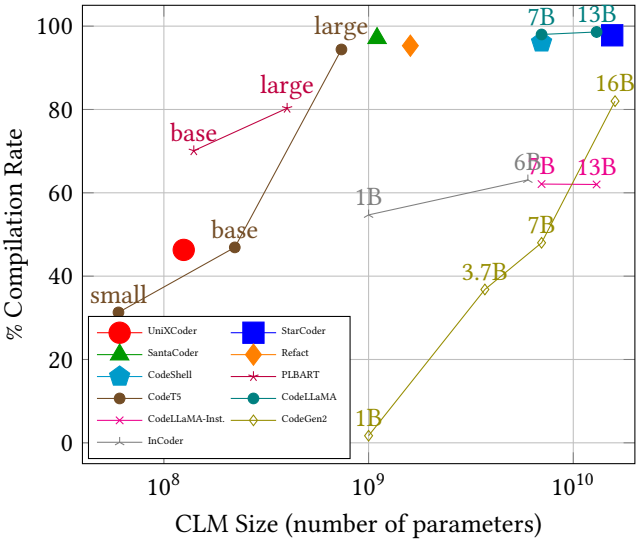


Figure 3. Compilation rates (%) for CLMs, averaged over 5 infills per model, evaluated across 163 infill tasks.

Summary of Results for RQ1

We found that nucleus sampling generally performs better than beam search in fixing bugs, though the differences are not statistically significant. There is also no clear advantage between sampling sizes 5 and 10 in beam search, but nucleus sampling's configuration significantly impacts results. We also observed a strong positive correlation between CLM's size and performance regarding bugs fixed and compiler errors, with CodeLLaMA-13B and StarCoder showing the best results. CodeGen2, however, was an exception across all its versions.

4.2 RQ2: Resource Usage of CLMs

This research question explores how the *memory* and *time* usage of different CLMs correlate with their bug-fixing performance. Since beam search and nucleus sampling have drastically different memory usage characteristics, it is not appropriate to compare their resource usage with each other. In this section, we compare the memory and time usage for nucleus sampling for all models except UniXCoder, as it is only run with beam search. Detailed memory and time usage for beam search for all models is available in [1].

4.2.1 Memory Usage: Fig. 4 shows the performance of the CLMs in relation to peak Video RAM (VRAM) used by the GPU, focusing on the best results for $N = 1$. The pattern is similar to those for CLM sizes depicted in Fig. 2a, except CodeGen2-1B, which uses more memory than its 3.7B, 7B, and 16B counterparts while generating far fewer correct infills.

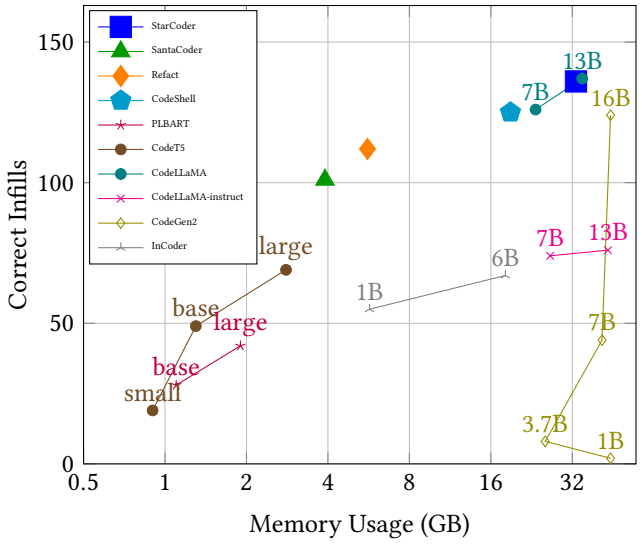


Figure 4. Performance of various CLMs versus peak VRAM usage for $N = 1$ in nucleus sampling.

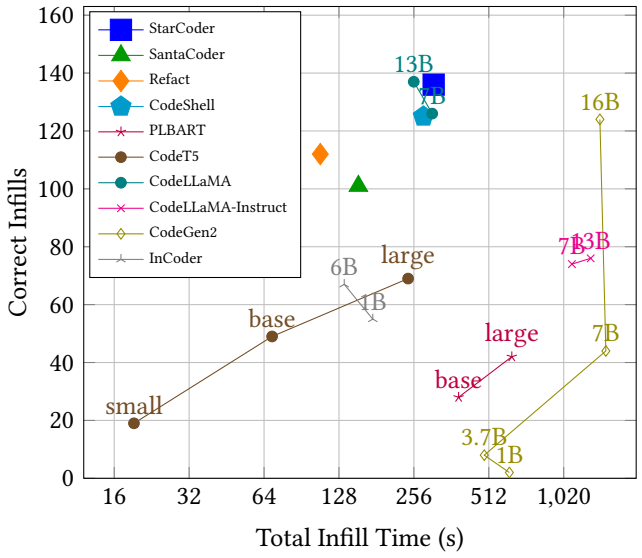


Figure 5. Performance of various CLMs versus infilling time for $N = 1$ in nucleus sampling.

Overall, CodeGen2 was the least memory-efficient and least effective at bug fixing compared to the other models. For all models except CodeGen2-1B, memory usage grew with increasing model size, and this relationship was found to be significantly positive in all configurations in nucleus sampling, and not solely for the best configuration, with a Spearman correlation coefficient of $rs = 0.823$ and a p-value of $p = 1.53 \times 10^{-5}$. Notably, a similar pattern was observed with beam search except it was more memory-intensive due to its exhaustive candidate exploration and storage requirements. On average, beam search required 22% more memory than nucleus sampling for the best results for $N = 1$. This difference was statistically significant, as confirmed by the Wilcoxon Signed-Rank Test with a p-value of 0.0005035.

4.2.2 Time Usage: Turning our attention to the infill time, we present the performance of CLMs compared to the GPU time required to generate infills for each task. Fig. 5 illustrates this comparison for best result for $N = 1$ using nucelus sampling. CodeT5-Small and Base are the most efficient, taking 19 and 69 seconds respectively, while CodeGen2 (7B and 16B), and CodeLLaMa-Instruct (7B and 13B) are the most time-consuming, each taking over 17 minutes.

In comparing model size and time, we observed four exceptions: InCoder 1B vs. 6B, CodeLLaMa 7B vs. 13B, CodeGen2 1B vs. 3.7B, and Santacode vs. Refact. In these cases, the larger model not only exhibited better infilling performance but also required less time to generate infills. Additionally, we observed that PLBART requires more time compared to most other models, despite being one of the smallest models with weaker performance. This pattern was consistent when averaging the generation time across all 28 configurations in nucleus sampling, not just the best configuration.

A similar pattern was observed with beam search, although it was more time-intensive than nucleus sampling. On average, beam search required 85% more time than nucleus sampling to achieve the best results for $N = 1$. This difference was statistically significant, as confirmed by the Wilcoxon Signed-Rank Test (p-value of 0.0001907).

Summary of Results for RQ2

We found that beam search requires more memory and time than nucleus sampling. While larger models show a positive correlation with memory usage, this does not hold for time usage. Additionally, CodeGen2's memory and time usage are largely unaffected by the number of parameters, but versions of CodeGen2 with fewer parameters tend to perform poorly.

4.3 RQ3: Diversity of Code Infills

This research question examines how code infill diversity varies across the used CLMs and what this implies for hybrid repair strategies. Table 3 presents a comparative analysis of bug distributions addressed by each CLM. For instance, PLBART-Base fixes 41 bugs from the dataset, but only 13 of those are fixed by CodeGen2-7B—which on its own fixes 44 bugs—indicating notable divergence in which bugs each model handles. UniXCoder fixes fewer bugs overall (26), yet shares 16 fixes in common with PLBART-Base, suggesting complementary strengths. Even two versions of PLBART display only about 50% overlap (27 common fixes) despite nearly the same correct infills for $N = 1$ (41 vs. 42).

Table 3. Comparative analysis of bug distributions addressed by each model at $N = 1$. Numbers in each cell indicate how many bugs *of those* fixed by the model from that *row*, were fixed by the model from that *column*.

	UniXCoder	Refact	SantaCoder	CodeShell	StarCoder	PLBART Base	PLBART Large	CodeT5 Small	CodeT5 Base	CodeT5 Large	CodeLLaMA 7B	CodeLLaMA 13B	CodeLLaMA Instr. 7B	CodeLLaMA Instr. 13B	CodeGen2 1B	CodeGen2 3.7B	CodeGen2 7B	CodeGen2 16B	InCoder 1B	InCoder 6B
UniXCoder	**26**	25	24	25	26	16	20	8	18	26	23	25	17	17	2	2	7	25	18	17
Refact		**112**	86	99	105	39	40	21	46	61	100	106	57	63	2	7	39	105	58	60
SantaCoder			**101**	90	97	37	39	21	44	59	90	94	53	57	2	7	38	94	58	61
CodeShell				**125**	120	38	41	21	48	66	112	119	66	68	2	7	43	112	59	65
StarCoder					**136**	40	42	22	49	68	121	130	71	74	2	8	44	121	63	70
PLBART Base						**41**	27	11	29	38	37	39	26	21	2	3	13	41	30	27
PLBART Large							**42**	9	27	34	39	41	27	26	2	3	16	42	29	27
CodeT5 Small								**22**	18	19	20	22	11	13	1	3	11	21	18	18
CodeT5 Base									**50**	47	47	49	27	33	2	4	22	49	37	37
CodeT5 Large										**69**	60	67	40	39	2	6	26	65	45	47
CodeLLaMA 7B											**126**	124	68	74	2	7	42	111	57	63
CodeLLaMA 13B												**137**	73	75	2	7	44	119	62	69
CodeLLaMA Instr. 7B													**74**	40	2	7	22	65	38	45
CodeLLaMA Instr. 13B														**76**	2	3	29	68	38	36
CodeGen2 1B															**2**	1	2	2	2	1
CodeGen2 3.7B																**8**	5	8	6	6
CodeGen2 7B																	**44**	41	25	29
CodeGen2 16B																		**124**	61	67
InCoder 1B																			**63**	50
InCoder 6B																				**70**

When comparing CodeLLaMA-13B and StarCoder—both fixing 137 and 136 bugs, respectively—we see an overlap of 130 fixes, yet each captures some bugs the other misses, and both still leave at least one PLBART-Base fix untouched. This underscores that individual CLMs bring distinct infill behaviors to the table.

We also observed 13 bugs that none of the models fixes, with 6 remaining unresolved even at $N = 5$. Interestingly, at $N = 5$, smaller models like UniXCoder and PLBART-Large each contribute two unique fixes, suggesting that including a diverse set of models—even those with lower standalone performance—can capture corner cases. To leverage these complementary strengths, industrial repair systems could combine multiple CLMs in parallel (to collect all candidate patches) and in sequence (where one model's output is validated or refined by another), thereby maximizing overall fix coverage.

This analysis shows that despite promising statistics, there are still relatively simple software bugs that contemporary pre-trained CLMs cannot fix automatically — which is about 8% even among very simple programs in HumanEval-Java [43]. These findings align with broader concerns discussed in recent literature [39], which emphasize that, despite their

power, CLMs are still limited in their ability to reason reliably about code. They often struggle to generalize beyond training patterns, reinforcing that CLMs presently serve as *data-driven assistants* rather than fully autonomous fix generators.

Building upon these insights, our evaluation results so far have mostly focused on determining the effectiveness of CLMs in generating the first infill. However, in many contexts, the ability to generate additional *unique* infills of high quality is just as crucial. For instance, newly proposed APR techniques can leverage additional infills in case the first infill is rejected. With beam search, diverse results are easily obtained, as over 4.5 average unique infills were observed for the most used CLMs. For example, SantaCoder averages 4.8 unique infills for beam sizes of 5. Conversely, the experiments that were conducted show that all CLMs produced fewer than two unique infills per task for most nucleus sampling configurations. Nevertheless, increasing top-p and temperature values in nucleus sampling can improve diversity.

We will now examine the trade-off between patch quality and diversity in nucleus sampling, using results from StarCoder as a reference. Similar patterns have been observed across most CLMs.

Fig. 6a illustrates the average number of unique infills generated by SantaCoder for $N = 5$ across different temperature and top-p settings. The diversity of outputs ranges from 1.0 at lower settings to 4.3 at the highest temperature (1.9) and top-p (0.8). This trend clearly demonstrates that increasing randomness leads to greater variability in generated results.

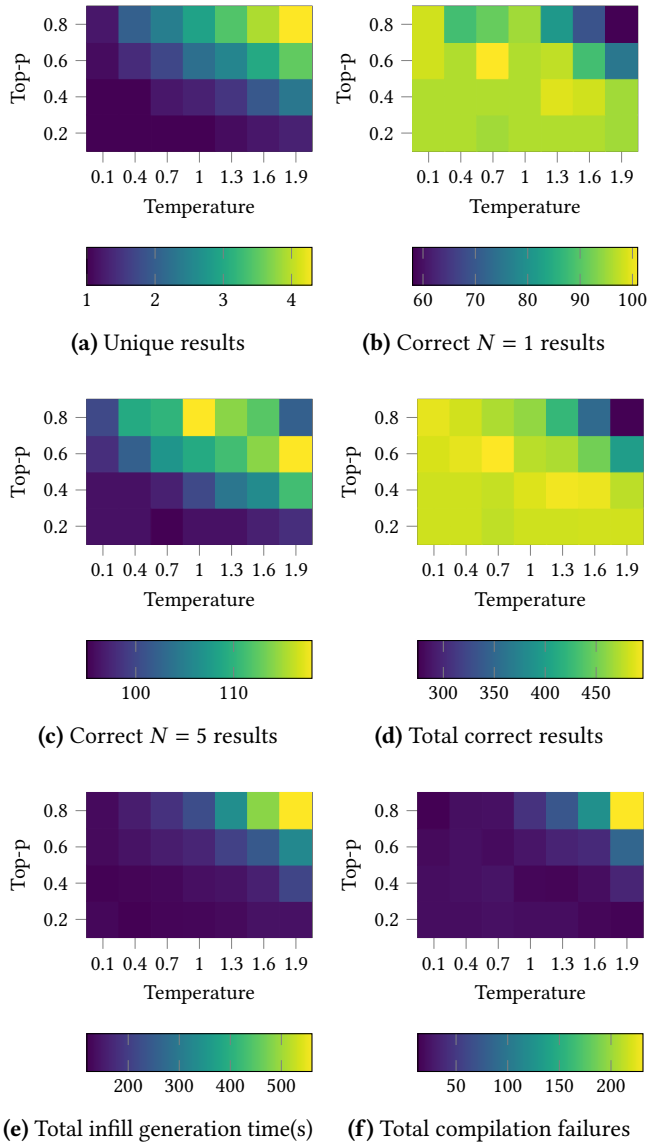
Fig. 6b and Fig. 6c illustrate the number of correct results for $N = 1$ and $N = 5$, respectively. The highest accuracy for $N = 1$ is achieved at (0.7, 0.6) with 101 correct results, while the lowest occurs at (1.9, 0.8) with only 58 correct results. For $N = 5$, accuracy improves at moderate settings, peaking at 118 correct results at (1.9, 0.6) before dropping sharply at (1.9, 0.8), where increased diversity leads to a decline in correctness.

Fig. 6d reinforces this trend, showing that total correct results decline as randomness increases beyond an optimal range. The highest accuracy, with 488 correct results per task, is observed at (0.7, 0.6), whereas the lowest, at 275 correct results, occurs at (1.9, 0.8), highlighting the trade-off between diversity and correctness.

Fig. 6e demonstrates that generation time rises significantly as diversity increases. The shortest total generation time, 119s, is recorded at (0.4, 0.2), whereas the longest, 560s, occurs at (1.9, 0.8), indicating that higher randomness comes with a substantial computational cost.

Fig. 6f shows that higher diversity settings lead to a significant rise in compilation failures. The failure count is lowest (13) at lower to moderate randomness settings but escalates sharply to 231 at high-temperature and high top-p configurations (1.9, 0.8), emphasizing the trade-off between diversity and output reliability.

Moderate settings (0.7, 0.6) offer the best balance of accuracy and diversity. While high-diversity configurations (1.9, 0.8) increase uniqueness (4.3 unique results per task), they also reduce correctness, extend computation time (up to 560s), and increase failures (up to 231). The trade-off is clear: greater randomness enhances variety but compromises reliability, making moderate settings the optimal choice.



(a) Unique results **(b)** Correct $N = 1$ results

(c) Correct $N = 5$ results **(d)** Total correct results

(e) Total infill generation time(s) **(f)** Total compilation failures

Figure 6. Heatmap of SantaCoder infilling metrics around result diversity.

Summary of Results for RQ3

Our analysis reveals that despite strong overall performance metrics, contemporary pre-trained CLMs still struggle with resolving certain simple bugs. More importantly, different models tend to fix different subsets of bugs, leading to comparable success rates but varying practical utility. This variation highlights the potential of combining models in hybrid repair strategies to enhance coverage.

The in-depth case study on StarCoder shows that increasing diversity through sampling (e.g., higher temperature and top-p) often comes at the cost of correctness, compilation reliability, and computation time. Therefore, applications must carefully balance diversity and quality based on their specific requirements.

Similar trade-offs were consistently observed across other high-performing CLMs—including Refact, CodeT5-Large, CodeLLaMA, and CodeShell—emphasizing the broader relevance of these findings for hybrid and adaptive repair systems.

5 Threats to Validity and Limitations

Our experiments face several threats to validity. First, we evaluate CLM performance using one and five generated infills, which suits our objectives but may not capture the full potential of CLMs in other contexts, such as integration with traditional bug-fixing methods. Second, each CLM was evaluated once per sampling configuration using a fixed seed. Running benchmarks with multiple seeds would improve our understanding of model stability, though at a higher computational cost. Third, our study evaluates a limited set of models, and as the field evolves, new models could yield different results. Fourth, we evaluated only a limited range of sampling parameters. While these parameters were selected based on preliminary testing, CLMs may perform better with parameters outside the employed range. A finer-grained exploration of parameter values might improve performance for certain models. Finally, while we use the HumanEval-Java benchmark, testing on additional datasets like Defects4J [45] could further validate our approach.

6 Conclusion and Future Work

This evaluation studies the effectiveness of the code infilling capabilities of 20 CLMs with various characteristics for automated program repair. The experiments show that effective CLMs exist in a range of sizes and can be selected based on computational resource budgets. Our results show that models like Refact and SantaCoder, with medium size, achieve high-quality bug fixes with acceptable resource consumption, while larger models like CodeLLaMA and StarCoder deliver superior performance but at a higher computational cost. These results underscore the importance of selecting the right CLM for specific APR scenarios, where trade-offs between repair quality, resource usage, and efficiency must be carefully considered.

We believe that diversity in produced repairs is a critical factor in advancing APR methods. Our findings suggest that the inherent variation in generated fixes can be leveraged by integrating CLMs with complementary approaches—such as search-based techniques—to enhance bug-fixing robustness. In this hybrid framework, CLMs are employed to generate a wide array of candidate patches, while search-based methods systematically refine, optimize, and combine these candidates to identify the most effective repair. Moreover, search-based techniques contribute a deeper, context-aware understanding of the surrounding context that compensates for the inherent limitations in CLMs' code knowledge. Our future research will focus on developing and rigorously evaluating this integrated approach. In addition, we plan to extend our evaluation to larger, more complex, real-world benchmarks—such as Defects4J [45]—to better assess the generalizability and effectiveness of our approach, and to broaden our exploration of CLM-based APR to encompass a wider range of programming languages beyond Java.

References

[1] May 2025. https://doi.org/10.5281/zenodo.15481569.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023). doi:10.48550/arXiv.2303.08774

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-Training for Program Understanding and Generation. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics. doi:10.18653/v1/2021.naacl-main.211

[4] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, et al. 2023. SantaCoder: Don't Reach for the Stars! *arXiv preprint arXiv:2301.03988* (2023). doi:10.48550/arXiv.2301.03988

[5] Dana Arad, Hadas Orgad, and Yonatan Belinkov. 2024. ReFACT: Updating Text-to-Image Models by Editing the Text Encoder. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. doi:10.18653/v1/2024.naacl-long.140

[6] Sidney Black, Stella Biderman, Eric Hallahan, Quentin Gregory Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. In *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*. doi:10.18653/v1/2022.bigscience-1.9

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021). doi:10.48550/arXiv.2107.03374

[8] CodeBERT-Base. Accessed: May 2025. https://huggingface.co/microsoft/codebert-base.

[9] CodeGen-16B. Accessed: May 2025. https://huggingface.co/Salesforce/codegen-16B-multi.

[10] CodeGen-2B. Accessed: May 2025. https://huggingface.co/Salesforce/codegen-2B-multi.

[11] CodeGen-350M. Accessed: May 2025. https://huggingface.co/Salesforce/codegen-350M-multi.

[12] CodeGen-6B. Accessed: May 2025. https://huggingface.co/Salesforce/codegen-6B-multi.

[13] CodeGen2-16B. Accessed: May 2025. https://huggingface.co/Salesforce/codegen2-16B_P.

[14] CodeGen2-1B. Accessed: May 2025. https://huggingface.co/Salesforce/codegen2-1B_P.

[15] CodeGen2-3.7B. Accessed: May 2025. https://huggingface.co/Salesforce/codegen2-3_7B_P.

[16] CodeGen2-7B. Accessed: May 2025. https://huggingface.co/Salesforce/codegen2-7B_P.

[17] CodeLlama-13b. Accessed: May 2025. https://huggingface.co/codellama/CodeLlama-13b-hf.

[18] CodeLlama-13b-Instruct. Accessed: May 2025. https://huggingface.co/codellama/CodeLlama-13b-Instruct-hf.

[19] CodeLlama-7b. Accessed: May 2025. https://huggingface.co/codellama/CodeLlama-7b-hf.

[20] CodeLlama-7b-Instruct. Accessed: May 2025. https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf.

[21] CodeShell-7B. Accessed: May 2025. https://huggingface.co/WisdomShell/CodeShell-7B.

[22] CodeT5-Base. Accessed: May 2025. https://huggingface.co/Salesforce/codet5-base.

[23] CodeT5-Large. Accessed: May 2025. https://huggingface.co/Salesforce/codet5-large.

[24] CodeT5-Small. Accessed: May 2025. https://huggingface.co/Salesforce/codet5-small.

[25] Sena Dikici and Turgay Tugay Bilgin. 2025. Advancements in Automated Program Repair: A Comprehensive Review. *Knowledge and Information Systems* (2025). doi:10.1007/s10115-025-02383-9

[26] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Future of Software Engineering*. IEEE. doi:10.1109/ICSE-FoSE59343.2023.00008

[27] Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. 2025. Patch Correctness Assessment: A Survey. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025). doi:10.1145/3702972

[28] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155* (2020).

[29] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. In *Proceedings of the 11th International Conference on Learning Representations*.

[30] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. doi:10.1145/3540250.3549098

[31] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019). doi:10.1145/3318162

[32] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-Training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022). doi:10.48550/arXiv.2203.03850

[33] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A Deep Dive into Large Language Models for Automated Bug Localization and Repair. *Proceedings of the ACM on Software Engineering* 1, FSE (2024). doi:10.1145/3660773

[34] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024). doi:10.1145/3695988

[35] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. doi:10.1109/ASE56229.2023.00181

[36] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. *arXiv preprint arXiv:2303.18184* (2023). doi:10.48550/arXiv.2303.18184

[37] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2024. Evolving Paradigms in Automated Program Repair: Taxonomy, Challenges, and Opportunities. *Comput. Surveys* 57, 2 (2024). doi:10.1145/3696450

[38] Hugging Face. Accessed: January 2025. https://huggingface.co/.

[39] Hans Hüttel. 2025. On Program Synthesis and Large Language Models. *Commun. ACM* 68, 1 (2025). doi:10.1145/3680410

[40] InCoder-1B. Accessed: May 2025. https://huggingface.co/facebook/incoder-1B.

[41] InCoder-6B. Accessed: May 2025. https://huggingface.co/facebook/incoder-6B.

[42] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of Experts. *arXiv preprint arXiv:2401.04088* (2024). doi:10.48550/arXiv.2401.04088

[43] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*. IEEE. doi:10.1109/ICSE48619.2023.00125

[44] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*. IEEE. doi:10.1109/ICSE48619.2023.00111

[45] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the ISSTA*. doi:10.1145/2610384.2628055

[46] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE. doi:10.1109/ICSE.2013.6606626

[47] Oleg Klimov and Sergey Vakhreev. 2023. Applying All Recent Innovations to Train a Code Model. https://refact.ai

[48] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2011). doi:10.1109/TSE.2011.104

[49] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: May the Source be with You! *arXiv preprint arXiv:2305.06161* (2023). doi:10.48550/arXiv.2305.06161

[50] Oebele Lijzenga, Iman Hemati Moghadam, and Vadim Zaytsev. 2025. Leveraging Search-Based and Pre-Trained Code Language Models for Automated Program Repair. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. ACM. doi:10.1145/3672608.3707774

[51] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair.

In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. doi:10.1145/3293882.3330577

[52] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018). doi:10.1145/3105906

[53] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report.

[54] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv preprint arXiv:2305.02309* (2023). doi:10.48550/arXiv.2305.02309

[55] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *Proceedings of the 11 International Conference on Learning Representations*.

[56] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. doi:10.1109/SP46215.2023.10179324

[57] PLBART-Base. Accessed: May 2025. https://huggingface.co/uclanlp/plbart-base.

[58] PLBART-Large. Accessed: May 2025. https://huggingface.co/uclanlp/plbart-large.

[59] Refact-1.6B. Accessed: May 2025. https://huggingface.co/smallcloudai/Refact-1_6B-fim.

[60] Joseph Renzullo, Pemma Reiter, Westley Weimer, and Stephanie Forrest. 2024. Automated Program Repair: Emerging trends pose and expose problems for benchmarks. *Comput. Surveys* (2024). doi:10.1145/3704997

[61] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code LlaMA: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023). doi:10.48550/arXiv.2308.12950

[62] SantaCoder-1.1B. Accessed: May 2025. https://huggingface.co/bigcode/santacoder.

[63] StarCoder-Base. Accessed: May 2025. https://huggingface.co/bigcode/starcoderbase.

[64] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (2019). doi:10.1145/3340544

[65] UnixCoder-Base. Accessed: May 2025. https://huggingface.co/microsoft/unixcoder-base.

[66] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* (2024). doi:10.1109/TSE.2024.3368208

[67] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. doi:10.18653/v1/2021.emnlp-main.685

[68] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. doi:10.1145/3597926.3598135

[69] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*. IEEE. doi:10.1109/ICSE48619.2023.00129

[70] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3540250.3549101

[71] Rui Xie, Zhengran Zeng, Zhuohao Yu, Chang Gao, Shikun Zhang, and Wei Ye. 2024. CodeShell Technical Report. *arXiv preprint arXiv:2403.15747* (2024). doi:10.48550/arXiv.2403.15747

[72] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (2018). doi:10.1109/TSE.2018.2874648

[73] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-Based Automated Program Repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023). doi:10.1145/3631974

[74] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, and Yun Yang Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. *arXiv preprint arXiv:2405.01466* (2024). doi:10.48550/arXiv.2405.01466

[75] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023. Pre-trained Model-based Automated Software Vulnerability Repair: How Far are We? *IEEE Transactions on Dependable and Secure Computing* (2023). doi:10.1109/TDSC.2023.3308897

[76] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. GAMMA: Revisiting Template-based Automated Program Repair via Mask Prediction. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. doi:10.1109/ASE56229.2023.00063

[77] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. *arXiv preprint arXiv:2311.07989* (2023). doi:10.48550/arXiv.2311.07989

[78] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM. doi:10.1145/3580305.3599790

[79] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* (2024). doi:10.1145/3708522

[80] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3468264.3468544

[81] Fida Zubair, Maryam Al-Hitmi, and Cagatay Catal. 2024. The Use of Large Language Models for Program Repair. *Computer Standards & Interfaces* (2024). doi:10.1016/j.csi.2024.103951

Received 2025-04-05; accepted 2025-05-19